USRS26
Cours 3 Compilation et débogage

Matthias Puech

Master 1 SEMS — Cnam

Compilation

Débogage

Compilation

Débogage

Compilation croisée

Générer un executable pour une *architecture* différente que celle sur laquelle le compilateur est executé.

Exemple

x86 sous Linux → ARM bare-metal

Architecture

- jeu d'instruction
- format d'exécutable
- ABI : interface entre le programme et le reste du système (conventions d'appel ; librairies ; accès à l'OS…)

Compilation croisée

Toolchain

L'ensemble des outils qui permettent de générer des programmes pour une architecture spécifique :

- pré-processeur
- compilateur
- assembleur
- éditeur de liens
- outils d'inspection, de traduction de fichiers binaires

Dans le monde GNU : `gcc` (compilateur) et `binutils` (le reste)

Compilation croisée

Toolchain

L'ensemble des outils qui permettent de générer des programmes pour une architecture spécifique :

- pré-processeur
- compilateur
- assembleur
- éditeur de liens
- outils d'inspection, de traduction de fichiers binaires

Dans le monde GNU : `gcc` (compilateur) et `binutils` (le reste)

Plus de détails l'année prochaine...

Vue d'ensemble

Le processus de compilation implique plusieurs acteurs :

moteur de production orchestre le processus (`make`)

préprocesseur C avec macros → C (`gpp`)

compilateur C → assembleur (`gcc`)

assembleur assembleur → fichier objet (`as`)

éditeur de lien fichier objet → executable symbolique (`ld`)

traducteur de format binaire executable symbolique → image binaire (`objcopy`)

`make` : le chef d'orchestre

`Makefile` : description *déclarative* du processus compilation.
(plus générique et incrémental que (p.ex.) un script shell.)

Un makefile est un ensemble de *règles* : une *cible* (fichier à générer), avec ses dépendances et la commande qui la crée.

Syntaxe

[cible] : *[dépendance]*...*[dépendance]*
[commande shell]

- *[cible]* est soit un fichier à créer, soit un nom
  (ex : `all` (par convention, compile "tout"),
  `clean` (par convention, nettoie l'arborescence des fichiers temporaires de compilation))
- *[dépendance]* est soit un fichier existant, soit une cible

Usage

`$ make` *[cible]*...*[cible]*
fabrique successivement chacune des *[cible]*s

make : le chef d'orchestre

Exemple (Cible générique)

"Pour fabriquer un `.o` à partir d'un `.s`, il faut utiliser `gcc -c` :"

```
%.o: %.s
	$(CC) -c $(ASFLAGS) $< -o $@
```

- `$(CC)` et `$(ASFLAGS)` sont des variables définies plus haut (`gcc`)
- `$@` est le nom de la cible (le fichier `.o`)
- `$<` est le nom de la (seule) dépendance (le fichier `.s`)

`make` : le chef d'orchestre

Exemple

Cible nommée "Si on me demande de faire `flash`, il faut d'abord faire `$(TARGET).bin` puis lancer `st-flash`"

```
flash: $(TARGET).bin
        st-flash erase
        st-flash write $< 0x8000000
```

`make` : le chef d'orchestre

Exemple

Cible nommée "Si on me demande de faire `flash`, il faut d'abord faire `$(TARGET).bin` puis lancer `st-flash`"

```
flash: $(TARGET).bin
        st-flash erase
        st-flash write $< 0x8000000
```

Processus récursive et incrémentale

- si une dépendance n'existe pas, `make` va rechercher si elle est la cible d'une règle, et la fabriquer (récursivement)
- si une cible est moins récente que ses dépendance, on la refabrique,
- si toutes les dépendances existent et sont à jour, on lance les commandes assocées

`make` : le chef d'orchestre

Exemple

Cible nommée "Si on me demande de faire `flash`, il faut d'abord faire `$(TARGET).bin` puis lancer `st-flash`"

```
flash: $(TARGET).bin
        st-flash erase
        st-flash write $< 0x8000000
```

Processus récursive et incrémentale

- si une dépendance n'existe pas, `make` va rechercher si elle est la cible d'une règle, et la fabriquer (récursivement)
- si une cible est moins récente que ses dépendance, on la refabrique,
- si toutes les dépendances existent et sont à jour, on lance les commandes assocées

Principe de la programmation logique (Prolog)

Pré-processing

Par commodité/convention, on utilise en C des *macros*.
Elles sont expansées par le pré-processeur gpp :

$$\text{texte} \xrightarrow{\text{gpp}} \text{texte}$$

- définir des constantes
  (ex : **#define** M_PI 3.14159276f)
- définir des “fonctions” dont la portée est dynamique et qui seront inlinées
  (ex : **#define** min(X, Y) ((X) < (Y) ? (X) : (Y)))
- inclure le contenu d’un fichier
  (ex : **#include** "leds.h")

Usage

```
$ gpp input.c > output.c
```

ou en utilisant `gcc` comme moteur :

```
$ gcc -E input.c -o output.c
```

Compilation (C → asm)

- “aplatit” le code C en du code assembleur
  (expressions, appels de fonctions, boucles, conditionnelles)
- donne une place mémoire à chaque variable et calcul intermédiaire (pile, variables globales, constantes)
- le code et les données sont réparties en *sections*
  (`.text`, `.data`, `.bss`...)

Compilation (C → asm)

- “aplatit” le code C en du code assembleur
  (expressions, appels de fonctions, boucles, conditionnelles)
- donne une place mémoire à chaque variable et calcul intermédiaire (pile, variables globales, constantes)
- le code et les données sont réparties en *sections*
  (`.text`, `.data`, `.bss`...)

Plongée dans `gcc` : les passes de compilation

- parsing du source (ASCII) en arbre de syntaxe abstraite
- traduction en code intermédiaire (pour `gcc` : SSA, RTL)
- optimisations (options `-f` et `-O`)
- émission du code assembleur

Compilation (C → asm)

- “aplatit” le code C en du code assembleur
  (expressions, appels de fonctions, boucles, conditionnelles)
- donne une place mémoire à chaque variable et calcul intermédiaire (pile, variables globales, constantes)
- le code et les données sont réparties en *sections*
  (`.text`, `.data`, `.bss`...)

Plongée dans `gcc` : les passes de compilation

- parsing du source (ASCII) en arbre de syntaxe abstraite
- traduction en code intermédiaire (pour `gcc` : SSA, RTL)
- optimisations (options `-f` et `-O`)
- émission du code assembleur

Usage

```
$ as fichier.s -o fichier.o
```

ou en utilisant `gcc` comme moteur :

```
$ gcc -S -o fichier.s fichier.c
```

Compilation (C → asm)

Quelques options de `gcc`

`-I` chemins où chercher les `#include`

`-mcpu=cortex-m4` choisit les instructions

`-mthumb` permet d'utiliser les instructions *thumb*
(plus concises, plus rapides)

`-mfpu/-mfloat*` génération d'instructions pour FPU
(ou émulation logicielle)

`-D`*VARIABLE* comme faire `#define` *VARIABLE*

`-O`n choisit le niveau d'optimisation ($n = 0..3$)

`-g` garde dans l'asm les symboles de déboggage
(backlinks de l'asm vers le source C)

Assembleur ARM, en deux mots

Structure d'un fichier asm

Chaque ligne est soit :

- une instruction (voir ci-dessous)
  (ex : "`ldr r3, =_sidata`")
- un label : une adresse mémoire à laquelle on donne un nom
  (ex : "`LoopFillZerobss:`")
- une directive :

  `.word` réserve la place pour un mot mémoire (32 bits)

  `.section` marque début de section (bloc contigu nommé)

Assembleur ARM, en deux mots

Quelques instructions

`ld*` `ri` *[addr]* lecture d'une adresse mémoire vers un registre

`st*` `ri` *[addr]* écriture d'un registre vers une adresse mémoire

`add*` `ri` `rj` additionne contenu de `ri` et `rj` et met resultat dans `ri`

`b*` *[label]* saute à *[label]* (toujours ou si condition)

Exemple : le code de démarrage

`lib/CMSIS/startup-stm32f303.s` dans les sources
lié avec le programme final

- point d'entrée du programme
  (l'instruction 0 = `Reset_Handler`)
- réserve une place pour `.isr_vector`, le vecteur d'interruption
- copie les données du programme de flash en mémoire
  (variables initialisées, variables `static`)
- initialise les variables non initialisées à 0
- appelle `SystemInit`
  (définie dans `system_stm32f3xx.c` : initialise le FPU, les horloges, les interruptions)
- appelle `main`
- ...boucle infinie

Assemblage (asm → objet *relocatable* ELF)

- traduit chaque instruction en son *opcode* (mot de 16/32 bits)
- calcule les adresses dans le code relativement aux adresses de début de section (*relocatable*)
- réserve la place pour les données déclarées
- propage les informations de déboggage
- exporte une liste de *symboles* (les points d'entrée)
  - les adresses exportées (*.word*)
  - les labels utilisés sans être définis

Usage

```
$ as -o fichier.o fichier.s
```

ou en utilisant `gcc` comme moteur (appelle `as`) :

```
$ gcc -c -o fichier.o fichier.s
```

Le format d'exécutable ELF

Une description symbolique de l'agencement de la mémoire agnostique à l'architecture. C'est le format des `.o` (relocatable ELF) et des `.elf` (executable ELF).

Structure

- en-tête (ABI, *relocatable* ou *executable*, point d'entrée, ...)
- liste de *sections* avec pour chacune :
  - nom
  - adresse mémoire et taille à l'exécution
  - contenu (code ou donnée)
- table de *symboles* :
  - nom
  - adresse mémoire ou UNDEFINED (pas de contenu)

Le format d'exécutable ELF

Usage

On lit les ELF avec `arm-none-eabi-readelf` et `arm-none-eabi-objdump`. Options :

`readelf -h` affiche l'en-tête

`objdump -h` la liste des sections

`readelf -s` la liste des symboles (alternative : `nm`)

`readelf -x` *n* dump le contenu en hexa de la section *n*

`objdump -d` désassemble les sections de code

Le format d'exécutable ELF

Usage

On lit les ELF avec `arm-none-eabi-readelf` et `arm-none-eabi-objdump`. Options :

`readelf -h` affiche l'en-tête

`objdump -h` la liste des sections

`readelf -s` la liste des symboles (alternative : `nm`)

`readelf -x` *n* dump le contenu en hexa de la section *n*

`objdump -d` désassemble les sections de code

Remarques

- ELF est le format standard des executables Linux (essayez `readelf -h /bin/ls`)
- format Mach-O pour macOS ; format PE pour Windows
- C'est l'OS qui interprète ELF et prépare la mémoire quand on lance un programme.

Édition des liens

Fusionne plusieurs objets ELF *relocatable* en un objet ELF *executable/absolu* :

- groupe le contenus des sections communes aux objets
  (toutes les sections `.text` sont concaténées)
- décide de l'emplacement mémoire final de chaque section
  (selon un *linker script*)
- fait la *relocation* : rend toutes les adresses absolues
- Résout tous les symboles UNDEFINED
  (s'il en reste, erreur : "`undefined reference to` *blah*")

Usage

```
$ ld -o f.elf -T linker_script.ld
    f1.o f2.o f3.o
```

Les sections

C'est une convention entre `gcc` et `ld` :

- `.text` le code source
- `.data` les variables globales initialisées
- `.rodata` les variables globales constantes, les chaînes de caractère
- `.bss` espace à allouer aux variables non initialisées
  (on donne juste la taille qu'il doit faire, pas son contenu)

Le linker script

`lib/CMSIS/STM32F303XC_FLASH.ld` dans les sources

- utilisé par `ld` pour produire l'executable `.elf`
- décrit comment sera organisé la mémoire à l'execution :
  - définit des *segments mémoire* (`FLASH`, `RAM`, ...)
  - affecte et ordonne les sections dans ces segments (`.text`, `.rodata` → `FLASH`; `.data`, `.bss` → `RAM`)
  - définit des constantes pour les débuts/fins de segments (`_sbss`, `_ebss`)
  - définit le point d'entrée du programme (instruction 0) (`Reset_Handler`)

⇝ un schéma de la mémoire d'un MCU particulier

Le format binaire brut `.bin`

Problème

Nous n'avons pas d'OS sur nos MCUs pour interpréter le ELF et préparer la mémoire.

Le format binaire brut `.bin`

Problème

Nous n'avons pas d'OS sur nos MCUs pour interpréter le ELF et préparer la mémoire.

⇝ `objcopy` va interpréter le ELF et préparer une image mémoire complètement statique.

Le format `.bin`

C'est le contenu exacte que doit avoir la mémoire si l'on veut que le MCU exécute le programme.

Usage

```
$ arm-none-eabi-objcopy -O binary main.elf main.bin
```

Remarques

Les raccourcis de gcc

```
# préprocessing, compilation et assemblage
$ gcc -c -o fichier.o fichier.c

# compilation, assemblage, edition des liens
$ gcc -o f.elf f1.c f2.c f3.s
```

Le paquet binutils

Tout ce qui n'est pas gcc :
as, ld, nm, strip, objcopy...

Compilation

Débogage

On-circuit debugging (OCD)

Dispositif de contrôle du MCU directement sur carte :

- contrôle de l'exécution (debogage de prototypes)
- lecture/écriture dans mémoires (pour charger le firmware)

Sur nos Discovery

Protocole *SWD*. Les pattes du MCU sont pré-câblées à l'interface SWD⟷USB intégrée (*ST-Link*)

Serial Wire Debug (SWD)

Interface électrique pour protocole JTAG. Deux modes :

Halt le processeur est arrêté

Monitor le processeur tourne normalement, mais s'arrête dès qu'une *condition* est rencontrée

Dans ces modes, commandes spéciales pour :

- lire/écrire dans registres et mémoire (→ Flash)
- poser une *condition* : écriture à adresse, valeur du registre PC (→ breakpoint)

Pinout

1. VDD (+3.3V)
2. SWCLK (horloge, de l'hôte à la cible)
3. GND
4. SWDIO (bus de données, bidirectionnel)
5. NRST (comme le bouton Reset)
6. SWO (?, fonctionnalité de trace avancées)

Lancer le serveur de débogage

L'outil `st-util`

- place le MCU en mode Halt
- attend connection d'un client sur port 4242
- execute commandes client

Usage

```
$ st-util -v
common.c: Loading device parameters....
common.c: Device connected is: F3 device, id 0x10036422
common.c: SRAM size: 0xa000 bytes (40 KiB), Flash: 0x40000
bytes (256 KiB) in pages of 2048 bytes
gdb-server.c: Chip ID is 00000422, Core ID is  2ba01477.
gdb-server.c: Listening at *:4242...
```

Lancer le serveur de débogage

L'outil `st-util`

- place le MCU en mode Halt
- attend connection d'un client sur port 4242
- execute commandes client

Usage

```
$ st-util -v
common.c: Loading device parameters....
common.c: Device connected is: F3 device, id 0x10036422
common.c: SRAM size: 0xa000 bytes (40 KiB), Flash: 0x40000
bytes (256 KiB) in pages of 2048 bytes
gdb-server.c: Chip ID is 00000422, Core ID is  2ba01477.
gdb-server.c: Listening at *:4242...
```

Alternative

`OpenOCD` : serveur plus complet et générique (et complexe).
Implémente JTAG pour nombreuses interfaces et cibles.

Lancer gdb

- envoie les commandes au serveur
- interface avec le code source (symboles de debug nécessaires)

Usage

```
$ arm-none-eabi-gdb main.elf
GNU gdb (GNU Tools for ARM Embedded Processors)
Reading symbols from main.elf...done.
(gdb) target remote localhost:4242
Remote debugging using localhost:4242
0x080003ec in SystemCoreClockUpdate () at
../lib/CMSIS/system_stm32f3xx.c:301
301 }
(gdb)
```

Lancer gdb

- envoie les commandes au serveur
- interface avec le code source (symboles de debug nécessaires)

Usage

```
$ arm-none-eabi-gdb main.elf
GNU gdb (GNU Tools for ARM Embedded Processors)
Reading symbols from main.elf...done.
(gdb) target remote localhost:4242
Remote debugging using localhost:4242
0x080003ec in SystemCoreClockUpdate () at
../lib/CMSIS/system_stm32f3xx.c:301
301 }
(gdb)
```

ou juste :

```
$ make debug
arm-none-eabi-gdb main.elf --eval-command="target
remote localhost:4242"
```

Commandes gdb

Le prompt

- `help` et la documentations de `gdb` sont vos amis
- toute commande peut être abrégée si elle n'est pas ambigue (ex : `lo` au lieu de `load`)
- ↑ et ↓ pour naviguer dans l'historique des commandes

1. "Contrôle-C"

Commandes gdb

Quelques commandes

`load`	charge le `.elf` en mémoire (flash)
`continue`	→ mode Monitor (exécution du code jusqu'à évenement)
`^C`[1]	→ mode Halt (stoppe l'exécution)
`step`	exécute une instruction *atomique* (e.g. `i++;`)
`next`	exécute une instruction (saute le code des appels de fonctions)
`print` *[e]*	affiche la valeur de l'expression *e* (y compris quand *e* est une variable en scope, ou un registre, e.g., `p` *pc*)
`set variable` *[v]* = *[e]*	affecte la valeur *e* à *v*.
`backtrace`	affiche la pile d'exécution (fonctions en cours d'exécution)

1. "Contrôle-C"

Commandes gdb

Quelques commandes (suite)

`stepi` exécute une instruction asm

`breakpoint` *[fichier]*`:`*[ligne]* → Halt dès qu'on passe sur une instruction correspondant à *[ligne]* dans *[fichier]*.

`watch` *[e]* → mode Halt dès que l'expression *e* change de valeur

`delete` enlève certains/tous les breakpoints

`(un)display` *[e]* (arrête d')affiche(r) la valeur de *e* à chaque fois qu'on s'arrête

`monitor reset` reset *to* mode Monitor avec possibilité d'entrer des commandes

`tui {en/dis}able` (dés)active l'affichage du code source, assembleur, l'état des registres

1. "Contrôle-C"